INFORMATION AND COMPUTATION 89, 1–34 (1990)

Type Inference for Polymorphic References

MADS TOFTE

Laboratory for Foundations of Computer Science,
Department of Computer Science,
University of Edinburgh,
Mayfield Road, Edinburgh EH9 3JZ, Scotland

The Hindley/Milner discipline for polymorphic type inference in functional programming languages is not sound if used on functions that can create and update references (pointers). We have found that the reason is a simple technical point concerning the capture of free type variables in store typings. We present a modified type inference system and prove its soundness using operational semantics. It is decidable whether, given an expression *e*, any type can be inferred for *e*. If some type can be inferred for *e* then a PRINCIPAL TYPE can be inferred. Principal types are found using unification. The ideas extend to polymorphic exceptions and have been adopted in the definition of the programming language STANDARD ML.

1. INTRODUCTION

It has been known for at least a decade that the Hindley/Milner type discipline for polymorphic type inference in functional programming languages is not sound if used on functions that can create assignable locations. (An example of a program that would type check but leads to a run-time type error will be shown below.) It has proved surprisingly difficult to understand precisely why this is so and to find a sound polymorphic type discipline.

The practical implication is that it has been hard to combine "imperative" language features such as references, assignment, arrays and even exceptions with the benefits of the Hindley/Milner polymorphism. The type discipline we shall present is identical to Milner's type discipline as far as purely applicative programs are concerned, but in addition it allows polymorphic use of references. The ideas extend to polymorphic exceptions and arrays. To give an example, we admit the following STANDARD ML program, which reverses lists in linear time,

```
fun fast_reverse(l)=
let val left = ref l and right = ref nil
```


0890-5401/90 $3.00

```
in  while !left <> nil do
      (right := hd(!left) :: !right; left := tl(!left));
    !right
end
```

Here the evaluation of `ref` *e* dynamically creates a new reference to the value of *e* and ! stands for dereferencing. Note that `fast_reserve` is an example of a polymorphic function, i.e. a function which can be applied to values of more than one type. Intuitively, the most general type of `fast_reverse` is $\forall t.t\ list \to t\ list$, where t ranges over all types.

1.1. *Related Work*

Hindley's type discipline (Hindley, 1969) uses type variables in type expressions. It has no quantification of type variables. Quantification of type variables plays a major role in Milner's system (Milner, 1978; Damas and Milner, 1982) because it is the quantification of type variables together with the related notion of instantiation that allows polymorphic use of functions defined by the user.

The problem of polymorphism and side effects is first described by Gordon, Milner, and Wadsworth (1979) in their definition of the first version of ML, which was used for the proof system LCF. They gave typing rules for so-called **letref** bound variables. (Like a PASCAL variable, a **letref** bound variable can be updated with an assignment operation but, unlike a PASCAL variable, a **letref** bound variable is bound to a permanent address in the store). The rules admitted some polymorphic functions that used local **letref** bound variables. Milner proved a soundness result using denotational semantics under the assumption that all assignments were monotyped; it was never proved that the rules for polymorphic use of **letref** bound variables were sound.

In his thesis Damas went further in allowing references as first-order values and he gave an impressive extension of the polymophic type discipline to cope with this situation (Damas, 1985). Damas correctly claimed that the problem with the unmodified type inference system is the rule for generalisation although he did not explain precisely why. He gave a soundness proof for his system; it was based on denotational semantics and involved a very difficult domain construction. Unfortunately, although his soundness theorem is not known to be false, there appears to be a fatal mistake in the soundness proof. (In his proof of Proposition 4, case **INST**, page 111, the requirements for using the induction hypothesis are not met.)

David MacQueen has developed yet another discipline for polymorphic references. It is currently implemented in the New Jersey ML compiler. More about this discipline will be said in the conclusion.

1.2. *Outline of the Paper*

The soundness proof we shall give for our type discipline differs from earlier proofs by being carried out in the setting of operational semantics instead of denotational semantics. This avoids the difficult domain construction. Instead, we use a simple, but very powerful proof technique concerning maximal fixed points of monotonic operators. Credit should go to Robin Milner for suggesting this absolutely crucial proof technique, which we call CO-INDUCTION (Milner and Tofte, 1990).

Thanks to this technique we can present two new results. First, we can actually pinpoint the problem in so far as we can explain precisely why the naive extension of the polymorphic type discipline is unsound. Second, we can present a new solution to the problem and prove it correct. Section 2 is devoted to presenting the first result. In Section 3 we present the type discipline together with examples of type inference and a type checking algorithm. The soundness is proved in Section 4. Completeness of the type checker (the existence of principal types) has been proved in detail, but the proof is too long to be included in this paper.

I assume (probably unjustly) that the reader has no prior knowledge of operational (relational) semantics, polymorphic type inference, and co-induction.

2. THE PROBLEM WITH POLYMORPHIC REFERENCES

The purpose of this section is to introduce basic notations and concepts and present the technical reason why type checking using the Milner discipline is not sound in the imperative setting.

To study the problem, we consider a minimal language, Exp, of expressions e obtained from the untyped lambda calculus by adding **let**. (As in ML we write **fn** x => e instead of $\lambda x.e$; **fn** is pronounced "lambda.") Here we assume a set Var of VARIABLES, ranged over by x,

$e ::=$	x	variable
|	**fn** x => e_1	lambda abstraction
|	$e_1 e_2$	application
|	**let** $x = e_1$ **in** e_2	let expression

The dynamic semantics is defined using a Plotkin style operational semantics (Plotkin, 1981). The basic idea is to write inference rules that allow us to infer conclusions of the form $s, E \vdash e \longrightarrow v, s'$, read: starting with store s and environment E, the expression e EVALUATES to value v and (a perhaps changed) store s'.

$$b \in \text{BasVal} = \textit{done}, \textit{true}, \textit{false}, 1, 2, \ldots,$$

$$v \in \text{Val} = \text{BasVal} + \text{Clos} + \{\textit{asg}, \textit{ref}, \textit{deref}\} + \text{Addr}$$

$$[x, e, E] \in \text{Clos} = \text{Var} \times \text{Exp} \times \text{Env}$$

$$s \in \text{Store} = \text{Addr} \xrightarrow{\text{fin}} \text{Val}$$

$$E \in \text{Env} = \text{Var} \xrightarrow{\text{fin}} \text{Val}$$

$$a \in \text{Addr}$$

FIG. 1. Objects in the dynamic semantics.

The semantic objects (basic values, values closures, stores, environments, and addresses) are defined in Fig. 1. Besides the booleans and the integers, the set of basic values contains the value *done* which is the value of expressions which are evaluated purely for the sake of their side effects. The values *ref*, *asg*, and *deref* are henceforth bound to the variables `ref`, `:=`, and `!`, respectively. We use the infix form $e_1 := e_2$ to mean $(:= e_1)\ e_2$. A lambda abstraction evaluates to a closure, consisting of the formal parameter x, the function body e, and an environment E, which gives the values of the free variables of the function.

Let A and B be sets. Then Fin A means the set of finite subsets of A. Moreover, $A + B$ means the disjoint union of sets, and $A \to^{\text{fin}} B$ means the set of finite maps from A to B (by a FINITE map we mean a function with finite domain). Any $f \in A \to^{\text{fin}} B$ can be written in the form $\{a_1 \mapsto b_1, \ldots, a_n \mapsto b_n\}$. In particular, the empty map is written $\{\ \}$. $\text{Dom}(f)$ means the domain of f. When f and g are (perhaps finite) maps then $f+g$, called f MODIFIED BY g, is the map with domain $\text{Dom}(f) \cup \text{Dom}(g)$ and values $(f+g)(a) = \text{if } a \in \text{Dom}(g) \text{ then } g(a) \text{ else } f(a)$. Note that $+$ is associative but not commutative.

The inference rules appear in Fig. 2. Every rule allows us from the premises above the line to conclude the conclusion below the line. For instance, rule 7 can be summarised as follows: if e_1 evaluates to v_1 in E and e_2 evaluates to v in E with x bound to v_1, then the let expression evaluates to v.

We shall write $\vdash e \longrightarrow v, s'$ for $\{\ \}, \{\ \} \vdash e \longrightarrow v, s'$.

2.1. *The Applicative Type Discipline*

We have to review Milner's polymorphic type discipline quite carefully in order to understand what goes wrong in the imperative case. We start with an infinite set, TyVar, of TYPE VARIABLES and a set, TyCon, of nullary TYPE CONSTRUCTORS,

$$\pi \in \text{TyCon} = \{\textit{int}, \textit{bool}, \ldots\}$$

$$\alpha \in \text{TyVar} = \{t, t', t_1, t_2, \ldots\}$$

$$\frac{x \in \mathrm{Dom}\, E}{s, E \vdash x \longrightarrow E(x), s} \tag{1}$$

$$\frac{}{s, E \vdash \mathbf{fn}\, x \Rightarrow e_1 \longrightarrow [x, e_1, E], s} \tag{2}$$

$$\frac{\begin{array}{l} s, E \vdash e_1 \longrightarrow [x_0, e_0, E_0], s_1 \\ s_1, E \vdash e_2 \longrightarrow v_2, s_2 \\ s_2, E_0 + \{x_0 \mapsto v_2\} \vdash e_0 \longrightarrow v, s' \end{array}}{s, E \vdash e_1 e_2 \longrightarrow v, s'} \tag{3}$$

$$\frac{\begin{array}{c} s, E \vdash e_1 \longrightarrow \mathit{asg}, s_1 \\ s_1, E \vdash e_2 \longrightarrow a, s_2 \\ s_2, E \vdash e_3 \longrightarrow v_3, s_3 \end{array}}{s, E \vdash (e_1 e_2)\, e_3 \longrightarrow \mathit{done}, s_3 + \{a \mapsto v_3\}} \tag{4}$$

$$\frac{s, E \vdash e_1 \longrightarrow \mathit{ref}, s_1 \quad s_1, E \vdash e_2 \longrightarrow v_2, s_2 \quad a \notin \mathrm{Dom}\, s_2}{s, E \vdash e_1 e_2 \longrightarrow a, s_2 + \{a \mapsto v_2\}} \tag{5}$$

$$\frac{s, E \vdash e_1 \longrightarrow \mathit{deref}, s_1 \quad s_1, E \vdash e_2 \longrightarrow a, s' \quad s'(a) = v}{s, E \vdash e_1 e_2 \longrightarrow v, s'} \tag{6}$$

$$\frac{s, E \vdash e_1 \longrightarrow v_1, s_1 \quad s_1, E + \{x \mapsto v_1\} \vdash e_2 \longrightarrow v, s'}{s, E \vdash \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2 \longrightarrow v, s'} \tag{7}$$

FIG. 2. Dynamic semantics.

Then the set of TYPES, Type, ranged over by τ and the set of TYPE SCHEMES, TypeScheme, ranged over by σ are defined by

$$\tau ::= \pi \mid \alpha \mid \tau_1 \to \tau_2$$

$$\sigma ::= \tau \mid \forall \alpha . \sigma_1 .$$

The arrow ($\to$) is right associative. Note that types contain no quantifiers and that type schemes contain outermost quantification only. This is necessary to get a type checking algorithm based on first-order term unification. A TYPE ENVIRONMENT is a finite map from program variables to type schemes:

$$TE \in \mathrm{TyEnv} = \mathrm{Var} \xrightarrow{\text{fin}} \mathrm{TypeScheme}.$$

A type scheme $\sigma = \forall \alpha_1 . \cdots . \forall \alpha_n . \tau$ is written $\forall \alpha_1 \cdots \alpha_n . \tau$. We say that $\alpha_1, \ldots, \alpha_n$ are BOUND in σ and that a type variable is FREE in σ if it occurs in τ and is not bound. Moreover, we say that a type variable is free in TE if it is free in a type scheme in the range of TE.

The map *tyvars*: Type $\to$ Fin(TyVar) maps every type τ to the set of type variables that occur in τ. More generally, *tyvars*(σ) and *tyvars*(TE) mean the set of type variables that occur free in σ and TE, respectively. Also, σ

and TE are said to be CLOSED if $tyvars\ \sigma = \varnothing$ and $tyvars\ TE = \varnothing$ and τ is said to be a MONOTYPE if $tyvars(\tau) = \varnothing$.

A SUBSTITUTION S is a map from type variables to types. It can be finite. By natural extension substitutions can be applied to types. This gives composition of substitutions with identity ID. As usual, $(S_2 \circ S_1)\,\tau$ means $S_2(S_1\tau)$, which we often write simply $S_2S_1\tau$.

The operation of putting $\forall\alpha.$ in front of a type or a type scheme is called GENERALISATION (ON α), or QUANTIFICATION (OF α), or simply BINDING (OF α). Conversely, τ' is an INSTANCE of $\sigma = \forall\alpha_1 \cdots \alpha_n.\tau$, written $\sigma \succ \tau'$, if there exists a finite substitution, S, with domain $\{\alpha_1, ..., \alpha_n\}$ and $S(\tau) = \tau'$. The operation of substituting types for *bound* type variables is called INSTANTIATION. Instantiation is extended to type schemes as follows: σ_2 is an INSTANCE of σ_1, written $\sigma_1 \geqslant \sigma_2$, if for all types τ, if $\sigma_2 \succ \tau$ then $\sigma_1 \succ \tau$. Write $\sigma_2 = \forall\beta_1 \cdots \beta_m.\tau_2$. One can prove that $\sigma_1 \geqslant \sigma_2$ if and only if $\sigma_1 \succ \tau_2$ and no β_j is free in σ_1. (This, in turn, is equivalent to demanding that $\sigma_1 \succ \tau_2$ and $tyvars(\sigma_1) \subseteq tyvars(\sigma_2)$). Finally, $\mathrm{Clos}_{TE}\tau$ means $\forall\alpha_1 \cdots \alpha_n.\tau$, where $\{\alpha_1, ..., \alpha_n\} = tyvars\ \tau \setminus tyvars\ TE$.

With this we can write down Milner's inference rules, see Fig. 3. The rules allow us to infer conclusions of the form $TE \vdash e \Rightarrow \tau$, read: e ELABORATES TO τ in environment TE. We refer to this type inference system as THE APPLICATIVE SYSTEM. (Readers familiar with the type inference system of Damas and Milner, 1982, will note that our version has neither an instantiation nor a generalisation rule. Instead instantiation is done precisely when variables are typed and generalisation is done explicitly by the closure operation in the let rule. Also note that the result of a typing is a type rather than a general type scheme. We claim without proof that the two systems admit exactly the same expressions. Our system has the advantage that whenever $TE \vdash e \Rightarrow \tau$, the form of e uniquely determines what rule was applied.)

We shall write $\vdash e \Rightarrow \tau$ for $\{\ \} \vdash e \Rightarrow \tau$.

$$\frac{x \in \mathrm{Dom}\ TE \quad TE(x) \succ \tau}{TE \vdash x \Rightarrow \tau}$$

$$\frac{TE + \{x \mapsto \tau'\} \vdash e_1 \Rightarrow \tau}{TE \vdash \mathbf{fn}\ x \Rightarrow e_1 \Rightarrow \tau' \to \tau}$$

$$\frac{TE \vdash e_1 \Rightarrow \tau' \to \tau \quad TE \vdash e_2 \Rightarrow \tau'}{TE \vdash e_1 e_2 \Rightarrow \tau}$$

$$\frac{TE \vdash e_1 \Rightarrow \tau_1 \quad TE + \{x \mapsto \mathrm{Clos}_{TE}\tau_1\} \vdash e_2 \Rightarrow \tau}{TE \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \Rightarrow \tau}$$

FIG. 3. The applicative type inference system.

2.2. *The Naive Extension and Why It Fails*

Let us first introduce a nullary type constructor, *stm*, and a unary postfix type constructor, *ref*:

$$\tau ::= \pi \mid \alpha \mid \tau_1 \to \tau_2 \mid stm \mid \tau\ ref. \tag{8}$$

(This extension induces an extension of the set of type schemes and type environments). The naive approach is to reuse the applicative system (Fig. 3) on the extended sets of semantic objects, with the additional requirement that the type environment bind **ref** to $\forall t.t \to t\ ref$, $:=$ *to* $\forall t.t\ ref \to t \to stm$, and ! to $\forall t.t\ ref \to t$.

However, with this system one can type unsafe programs. Consider, for example, the simple program

$$\textbf{let}\ \mathtt{r} = \mathtt{ref}(\textbf{fn}\ \mathtt{x} \Rightarrow \mathtt{x})\ \textbf{in}\ (\mathtt{r} := (\textbf{fn}\ \mathtt{x} \Rightarrow \mathtt{x}+\mathtt{1});\ !\mathtt{r}\ \mathtt{true}), \tag{9}$$

where ; stands for sequential evaluation (the dynamic and static inference rules for ; are unproblematic).

Although this program would lead to a run-time error, if run, it can be typed in the applicative discipline as follows. The expression `ref(`**fn** `x` $\Rightarrow$ `x)` can get type $(t \to t)\ ref$ and the body of the let expression is typable under the assumption $\{\mathtt{r} \mapsto \forall t.((t \to t)\ ref)\}$ using the instantiations $\forall t.((t \to t)\ ref) \succ (int \to int)\ ref$ and $\forall t.((t \to t)\ ref) \succ (bool \to bool)\ ref$ for the two occurrences of `r`.

The possibility of run-time errors in apparently well-typed programs is a consequence of a more fundamental inconsistency between the elaboration and the evaluation: an expression e can elaborate to a type τ and evaluate to a value v without v necessarily having type τ. For example, if we erase "`true`" from (9) then we get an expression which elaborates to the type $bool \to bool$ (among others), but the computed value, namely the successor function, is not of type $bool \to bool$.

One cannot help being sceptical about the way type variables are generalised and instantiated in the above example. To examine this matter more carefully, it is worth reflecting on why generalisation and instantiation are sound in the purely applicative setting. (That the applicative system is sound is a non-trivial fact; the purpose of the present informal discussion is merely to prepare the ground for the formal treatment.)

Consider the rule for elaboration of **let** expressions in Fig. 3 together with the evaluation rule

$$\frac{E \vdash e_1 \longrightarrow v_1 \quad E + \{x \mapsto v_1\} \vdash e_2 \longrightarrow v}{E \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \longrightarrow v}$$

Having elaborated e_1 to τ_1 in TE, we quantify all the type variables that occur free in τ_1 but not free in TE to obtain a type scheme σ; the free

occurences of x in e_2 can now elaborate to different types as long as these all are instances of σ. The soundness of the type inference system can be formulated as a consistency property of the static and the dynamic inference systems. In general, if the values in E have the types prescribed in TE and $TE \vdash e \Rightarrow \tau$ and $E \vdash e \longrightarrow v$ then we expect v to have type τ. In particular, for the **let** rules, we expect v_1 to have type τ_1. But why does v_1 have the more general type scheme $\mathrm{Clos}_{TE}\,\tau_1$? Let t be a type variable in τ_1. If t occurs free in TE, i.e., if t is free in $TE(y)$ for some y, then the type of v_1 depends on the type of the value $E(y)$, so we cannot generalise on t. On the other hand, assume that t does not occur free in TE. Then t is not determined by the type of any of the values that are bound to the free variables of e_1. Now it is a pleasant fact about purely functional languages that all one needs to know in order to give a type to a value v resulting from the evaluation of an expression e is the types of the values of the variables that occur free in e. Therefore, when t does not occur free in any of these types, we can generalise it.

The situation is slightly more involved when we extend the language with references. As before, if t occurs free in TE, it does not make sense to generalise it. But assume that t is not free in TE. In ascribing a type to the value v_1 it is no longer sufficient to know the types of the values of the free variables of e_1. The problem is that the evaluation $s, E \vdash e_1 \longrightarrow v_1, s_1$ may have created a new reference to a value (for instance v_1 itself) whose type contains t free. In this case, generalising t would destroy the connection between the types of the values that are stored and the types of the values that are the results of expressions.

To put these informal comments on firm ground, we shall now follow the style of (Lakatos, 1976) and try to prove a soundness theorem for the naive extension to see where the argument breaks down. Let us develop a little sequence of soundness propositions, starting from a very crude one. Each soundness proposition leads to the subsequent soundness proposition till we reach a proposition the proof of which fails because of just one interesting technical detail. This last soundness proposition is very useful, for it will become true once we have mended the type inference system.

Let us assume given a basic relation $\mathrm{IsOf} \subseteq \mathrm{BasVal} \times \mathrm{TyCon}$ relating basic values and nullary type constructors so that *true* IsOf *bool*, 3 IsOf *int*, etc. Let e be an expression, b be a basic value and π a nullary type constructor (such as *int* or *bool*).

FIRST SOUNDNESS PROPOSITION. *If* $\vdash e \Rightarrow \pi$ *and* $\vdash e \longrightarrow b, s'$ *then* b IsOf π.

An evaluation which produces a basic value can involve evaluations which produce nonbasic values such as closures and addresses about which the first proposition has nothing to say. It is clear, therefore, that we have

to extend the IsOf relation to a relation on Val × Type. Let $v:\tau$ be some extension of the IsOf relation (of course, not any extension will do—the analysis below will reveal some properties the : relation must have).

SECOND SOUNDNESS PROPOSITION. *If* $\vdash e \Rightarrow \tau$ *and* $\vdash e \longrightarrow v, s'$ *then* $v:\tau$.

In the first soundness proposition the resulting store (s') plays no role for the conclusion b IsOf π, because the store is of no importance to the typing of basic values. Not so in the second soundness proposition, where v can be an address. (Obviously, if v is an address a then the type of v depends on what s' contains at address a.) Thus, instead of looking for a binary relation $v:\tau$, it is natural to look for a ternary relation $s \models v:\tau$, read "given the store s, v has type τ."

THIRD SOUNDNESS PROPOSITION. *If* $\vdash e \Rightarrow \tau$ *and* $\vdash e \longrightarrow v, s'$ *then* $s' \models v:\tau$.

Now $\vdash e \Rightarrow \tau$ only if e contains no free variables. However, both the elaboration and the evaluation of e can involve expressions with free variables. Similarly, evaluations starting in the empty store may involve subcomputations that start in a non-empty store. To strengthen the third soundness proposition, first extend the $s \models v:\tau$ relation to a relation between stores, values, and type schemes by defining that $s \models v:\sigma$ if for all $\tau \prec \sigma$, $s \models v:\tau$. Then extend this relation to a relation between stores, environments, and type environments by pointwise extension: $s \models E: TE$ if Dom E = Dom TE and for all $x \in$ Dom E, $s \models E(x) : TE(x)$.

FOURTH SOUNDNESS PROPOSITION. *If* $s \models E: TE$ *and* $TE \vdash e \Rightarrow \tau$ *and* $s, E \vdash e \longrightarrow v, s'$ *then* $s' \models v:\tau$.

With any sensible definition of the $s \models v:\tau$ relation, this proposition is false. To see this, consider the following example: Let

$$s = \{\mathrm{a} \mapsto \mathrm{nil}\}$$

$$E = \{\mathrm{x} \mapsto \mathrm{a}, \mathrm{y} \mapsto \mathrm{a}\}$$

$$TE = \{\mathrm{x} \mapsto (\textit{int list})\ \textit{ref}, \mathrm{y} \mapsto (\textit{bool list})\ \textit{ref}\}$$

$$e = (\mathrm{x} := [7]);\ !\mathrm{y}$$

where e can be regarded as syntactic sugar for (**fn** z => !y)(x := [7]). Notice that x and y are bound to the same address. We have $s \models E: TE$ because x is bound to a and s(a) has type *int list* and, similarly, y is bound to a and s(a) has type *bool list*. Moreover, we have $TE \vdash e \Rightarrow$ *bool list* and $s, E \vdash e \longrightarrow [7], s'$, but certainly not $s' \models [7]$: *bool list*. Therefore, we have a counterexample to the fourth soundness proposition.

From this counterexample we learn that the typing of values depends on not only the dynamic store but also on a particular *typing* of the store. Let us define a STORE TYPING, ST, to be a map from addresses to types, and let us assume that we can define a quaternary relation $s: ST \models v: \tau$, read "given the typed store $s: ST$, the value v HAS TYPE τ," where a TYPED STORE is a pair (s, ST) such that $\mathrm{Dom}(s) = \mathrm{Dom}(ST)$. As before, such a relation can be extended to a relation $s: ST \models v: \sigma$, which in turn can be extended to a relation $s: ST \models E: TE$, read "E MATCHES TE given the typed store $s: ST$."

FINAL SOUNDNESS PROPOSITION. *If* $s: ST \models E: TE$ *and* $TE \vdash e \Rightarrow \tau$ *and* $s, E \vdash e \longrightarrow v, s'$ *then there exists a store typing* ST' *such that* $s': ST' \models v: \tau$.

Notice that a store typing maps addresses to types, not type schemes, thus preventing stored objects from having quantified polymorphic types (although there can be free type variables in the store typing). Having general type schemes in store typings turns out to undermine the theory of type inference and principal types, see Section 4.2.

Given that store typings map addresses to types, we expect it to be the case that

$$s: ST \models a: \tau \qquad \text{if and only if}$$

$$\tau = (ST(a))\ ref \text{ and } s: ST \models s(a): ST(a). \tag{10}$$

If we want to be able to prove the final soundness proposition, it will not suffice to have store typings map addresses to monotypes; for example, if $s = ST = \{\ \}$ and $\{\ \}: \{\ \} \models E: TE$ and $e = \texttt{ref}(\textbf{fn}\ \texttt{x} \Rightarrow \texttt{x})$, we have $TE \vdash e \Rightarrow (t \to t)\ ref$ and $\{\ \}, E \vdash e \longrightarrow a, \{a \mapsto [\texttt{x}, \texttt{x}, E]\}$, for some a. Therefore, if we are to obtain the conclusion of the final proposition, i.e.,

$$\{a \mapsto [\texttt{x}, \texttt{x}, E]\}: ST' \models a: (t \to t)\ ref$$

then $ST'(a)$ must be $t \to t$, c.f., (10) i.e., ST' is an example of a store typing in which a type variable occurs free. This is why we let store typings map adresses to types.

The type variables that occur in store typings are extremely important. In fact, they reveal what goes wrong in unsound inferences, as should soon become clear.

In order to attempt to prove the final soundness proposition one first has to define the typing relation $s: ST \models v: \tau$. I ask the reader to believe that there is a definition of the typing relation such that the fixed point equation (10) holds and such that if one attempts to prove the final soundness proposition by induction on the depth of inference of $s, E \vdash e \longrightarrow v, s'$ then all the cases go through, except one, namely the case concerning **let**

expressions. We shall now see that the proof of the **let** case breaks down in the most illuminating way which gives a hint for how to improve the type inference system. Let us assume that we have dealt successfully with all other cases; we then come to the dynamic inference rule for let expressions:

$$\frac{s, E \vdash e_1 \longrightarrow v_1, s_1 \quad s_1, E + \{x \mapsto v_1\} \vdash e_2 \longrightarrow v, s'}{s, E \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \longrightarrow v, s'}. \tag{11}$$

The conclusion $TE \vdash e \Rightarrow \tau$ must have been by the rule

$$\frac{TE \vdash e_1 \Rightarrow \tau_1 \quad TE + \{x \mapsto \mathrm{Clos}_{TE}\tau_1\} \vdash e_2 \Rightarrow \tau}{TE \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \Rightarrow \tau}. \tag{12}$$

(Recall that $\mathrm{Clos}_{TE}\tau_1$ means $\forall\alpha_1 \cdots \alpha_n.\tau_1$, where $\{\alpha_1, ..., \alpha_n\}$ are the type variables in τ_1 that are *not* free in TE.)

We now apply the induction hypothesis to the first premise of (11) together with the first premise of (12) and the given $s: ST \models E: TE$. Thus there exists a ST_1 such that

$$s_1: ST_1 \models v_1: \tau_1. \tag{13}$$

Before we can apply the induction hypothesis to the second premise of (11), we must establish $s_1: ST_1 \models E + \{x \mapsto v_1\}: TE + \{x \mapsto \mathrm{Clos}_{TE}\tau_1\}$ and to get this, we must strengthen (13) to

$$s_1: ST_1 \models v_1: \mathrm{Clos}_{TE}\tau_1. \tag{14}$$

It is precisely this step that goes wrong, if by taking the closure we generalise on type variables that occur free in ST_1. The snag is that when we have imperative features, there are really *two* places a type variable can occur free, namely (1) the type environment and (2) the store typing. In both cases, generalisation on such a type variable is wrong.

> *The naive extension of the polymorphic type discipline fails because it admits generalisation on type variables that occur free in the store typing.*

The unsafe program (9) gives a concrete illustration of this point. Assuming $s = ST = \{\ \}$, the evaluation is

$$\{\ \}, E \vdash \texttt{ref}(\textbf{fn } \texttt{x} \Rightarrow \texttt{x}) \longrightarrow a, \{a \mapsto [\texttt{x}, \texttt{x}, E]\}$$

and the elaboration $TE \vdash \mathrm{ref}(\textbf{fn } \mathrm{x} \Rightarrow \mathrm{x}) \Rightarrow (t \to t)\ ref$. Assuming $\{\ \}: \{\ \} \models E: TE$, the induction hypothesis yields an ST_1 such that

$$\{a \mapsto [\mathrm{x}, \mathrm{x}, E]\}: ST_1 \models a: (t \to t)\ ref \tag{15}$$

from which it follows that $ST_1(a)$ must be $t \to t$. The free occurrence of t in ST_1 expresses a dependence of the type of a on the store typing. Therefore, we cannot strengthen (15) to

$$\{a \mapsto [\mathrm{x}, \mathrm{x}, E]\} : \{a \mapsto (t \to t)\ ref\} \models a : \forall t.(t \to t)\ ref.$$

Unfortunately, store typings cannot be included in the sentences of the type inference system since not even the domain of the store is known at compile time. Instead one can enrich the sentences in other ways to give perhaps conservative, but at least safe, approximations of the set of type variables that would occur in the store typing. In effect, Damas' system (Damas, 1985), the system we now present (Tofte, 1988) and David MacQueen's system can all be seen as taking this approach. A different approach was taken in the original ML. Here references were barred from being values, thus making the use of a reference more syntactically obvious, but even so it was necessary to give additional contraints to ensure that references that are embedded in closures ("own" variables), and hence can escape their scope, are monomorphic—see (Gordon, Milner, and Wadsworth, 1979, p. 49, rule (2)(i)(b)) for details.

3. The Imperative Type Discipline

We first present the type inference system. Then we give examples of its use and present a type checker.

3.1. *The Inference System*

The basis idea is to modify the type expressions so that there is a visible difference between those types that occur in the implicit store typing and those that do not. This can be achieved by having two disjoint sets of type variables; ImpTyVar is the set of IMPERATIVE type variables and AppTyVar is the set of APPLICATIVE type variables:

$t \in \mathrm{AppTyVar} = \{t, t_1, ...,\}$	applicative type variables
$u \in \mathrm{ImpTyVar} = \{u, u_1, ...,\}$	imperative type variables
$\alpha \in \mathrm{TyVar} = \mathrm{AppTyVar} \cup \mathrm{ImpTyVar}$	type variables.

The applicative type variables are called applicative because they correspond exactly to the type variables in the applicative type discipline. The imperative type variables are called imperative because they only are needed in imperative languages; they range over types of values that

(perhaps) occur in the store. Types are defined by (8), where α now ranges over both imperative and applicative type variables. The set of IMPERATIVE types, ranged over by θ, is the set of types that contain no applicative type variables. For a value to stored it must have an imperative type. Type schemes and type environments are defined as before, except of course that now each bound variable is either applicative or imperative. When T is a type, a type scheme, or a type environment then *tyvars*(T) means all the type variables that occur free in T while *apptyvars* T means all the applicative type variables that occur free in T.

A SUBSTITUTION S is now a map from type variables to types which maps imperative type variables to imperative types. (Hence the image of an imperative type variable cannot contain applicative type variables, but the image of an applicative type variable can contain imperative type variables.) The definition of instantiation, $\sigma \succ \tau$, is as before but now with the new meaning of substitution.

In addition to $\text{Clos}_{TE}\tau$ defined earlier, we now define $\text{AppClos}_{TE}\tau$ to mean $\forall\alpha_1 \cdots \alpha_n . \tau$, where $\{\alpha_1, ..., \alpha_n\} = apptyvars\ \tau \backslash apptyvars\ TE$ is the set of all *applicative* type variables in τ not free in TE.

An expression is said to be NON-EXPANSIVE if it is a variable or a lambda abstraction. All other expressions, i.e., applications and let expressions, are said to be EXPANSIVE. Although this distinction is purely syntactical it is supposed to suggest the dynamic behaviour; the dynamic evaluation of a non-expansive expression cannot expand the domain of the store, while the evaluation of an expansive expression might. Our syntactic classification is very crude as there are many expansive expressions that in fact will not expand the domain of the store. The classification is chosen so as to be very easy to remember; the proofs that follow do not rely heavily on this very crude classification.

The type inference rules appear in Fig. 4 and they allow us to infer sentences of the form $TE \vdash e \Rightarrow \tau$. We see that the first three rules are as before but that the let rule has been split into two rules. In (20), where e_1 is expansive, an imperative type variable u in τ_1 is a warning that u may occur in the type of a reference created during the evaluation of e_1. By closing applicative type variables only, we avoid generalisation on u; the applicative type variables in τ_1 cannot occur in the types of values in the store as all stored values must have imperative types. In (19), where e_1 is non-expansive, no new reference can be created by e_1 so there is no need to distinguish between imperative and applicative type variables when closing τ_1.

Notice that if TE contains no imperative type variables (free or bound) then every type inference that could be done in the original system can also be done in the new system, using applicative type variables only; in rule (20) when τ_1 contains no imperative type variables then taking the

$$\frac{x \in \mathrm{Dom}\, TE \quad TE(x) \succ \tau}{TE \vdash x \Rightarrow \tau} \tag{16}$$

$$\frac{TE + \{x \mapsto \tau'\} \vdash e_1 \Rightarrow \tau}{TE \vdash \textbf{fn}\, x \Rightarrow e_1 \Rightarrow \tau' \to \tau} \tag{17}$$

$$\frac{TE \vdash e_1 \Rightarrow \tau' \to \tau \quad TE \vdash e_2 \Rightarrow \tau'}{TE \vdash e_1 e_2 \Rightarrow \tau} \tag{18}$$

$$\frac{e_1 \text{ is non-expansive} \quad TE \vdash e_1 \Rightarrow \tau_1 \quad TE + \{x \mapsto \mathrm{Clos}_{TE}\tau_1\} \vdash e_2 \Rightarrow \tau}{TE \vdash \textbf{let}\, x = e_1 \,\textbf{in}\, e_2 \Rightarrow \tau} \tag{19}$$

$$\frac{e_1 \text{ is expansive} \quad TE \vdash e_1 \Rightarrow \tau_1 \quad TE + \{x \mapsto \mathrm{AppClos}_{TE}\tau_1\} \vdash e_2 \Rightarrow \tau}{TE \vdash \textbf{let}\, x = e_1 \,\textbf{in}\, e_2 \Rightarrow \tau} \tag{20}$$

FIG. 4. The imperative type inference system.

applicative closure is the same as taking the ordinary closure. But in general TE will contain imperative type variables, as we shall assume

$$TE(\texttt{ref}) = \forall u.u \to u\ ref$$
$$TE(:=) = \forall t.t\ ref \to t \to stm$$
$$TE(!) = \forall t.t\ ref \to t.$$

Only the type of `ref` contains an imperative type variable, for only `ref` can create a new reference.

3.2. *Examples of Type Inference*

We first illustrate the difference between rules (19) and (20).

EXAMPLE 3.1. We have $TE \vdash \textbf{fn}\ \texttt{x} \Rightarrow\ !(\texttt{ref x}) \Rightarrow u \to u$ although not $TE \vdash \textbf{fn}\ \texttt{x} \Rightarrow\ !(\texttt{ref x}) \Rightarrow t \to t$. Still, we can type

$$\textbf{let}\ \texttt{f} = \textbf{fn}\ \texttt{x} \Rightarrow\ !(\texttt{ref x})\ \textbf{in}\ (\texttt{f(7); f(true)})$$

using the let rule for non-expansive expressions (rule (19)), which will allow a generalisation from $u \to u$ to $\forall u.u \to u$ in the type of `f`.

EXAMPLE 3.2. We have $TE \vdash \texttt{ref}(\textbf{fn}\ \texttt{x} \Rightarrow \texttt{x}) \Rightarrow (u \to u)\ ref$ but not $TE \vdash \texttt{ref}(\textbf{fn}\ \texttt{x} \Rightarrow \texttt{x}) \Rightarrow (t \to t)\ ref$. Consequently, in an expression of the form

$$\textbf{let}\ \texttt{r} = \texttt{ref}(\textbf{fn}\ \texttt{x} \Rightarrow \texttt{x})\ \textbf{in} \cdots$$

the let rule for expansive expressions, rule (20), will prohibit generalisation from $(u \to u)$ *ref* to $\forall u.((u \to u))$ *ref*). Thus the unsafe expression

let r = ref(**fn** x => x) **in** (r := (**fn** x => x + 1); ! r true)

from the previous section cannot be typed. Note, however, that

let r = ref(**fn** x => x) **in** (r := (**fn** x => x + 1); ! r 1)

is typable using $TE \vdash$ ref(**fn** x => x) $\Rightarrow$ (*int* $\to$ *int*) *ref* and rule 20.

For the remaining examples, let us temporarily extend the types with the unary type constructor *list* and assume that the type environment binds the variables nil, :: (infix construction of lists), hd and tl to the obvious polymorphic types involving applicative type variables only. The introduction of **while** loops into the language is straightforward.

EXAMPLE 3.3. Here is the fast_reverse function once again.

```
e1 = fn l =>
    let data = ref l in
    let result = ref nil in
      (while !data <> nil do
         (result := hd(!data) :: !result; data := tl(!data));
       ! result
      )
```

We have $TE \vdash e_1 \Rightarrow u\ list \to u\ list$; in the body of the second **let**, the type environment maps data to *u list ref* and result to *u list ref*. Notice that *u* cannot be generalised since *u* becomes free in the type environment at **fn** l =>. Now

```
let fast_reverse = e1
in (fast_reverse [1, 9, 7, 5]; fast_reverse [true, false,
false])
```

is typable using rule (19), which allows the generalisation from *u list* $\to$ *u list* to $\forall u.u\ list \to u\ list$.

As one would expect, since fast_reverse has type $\forall u.u\ list \to u\ list$ while the applicative reverse function has type $\forall t.t\ list \to t\ list$, there are programs that are typable with the applicative version only. One example is

```
let fast_reverse = ···
in let f = hd(fast_reverse[fn x => x])
   in (f(7); f(true))
```

EXAMPLE 3.4. This example illustrates what I believe to be the only interesting limitation of the inference system and how to get around it in

practice. The `fast_reverse` function is a special case of folding a function `f` (e.g., `cons`) over a list `l` starting with initial result `i` (e.g., `nil`):

$e_1 =$ **fn** `f` $\Rightarrow$ **fn** `i` $\Rightarrow$ **fn** `l` $\Rightarrow$
  let `data = ref l` **in**
  let `result = ref i` **in**
    (**while** `!data <> nil` **do**
      `(result:=f(hd(!data))(!result); data:=tl(!data));`
     `!result`
)

We have $TE \vdash e_1 \Rightarrow (u_1 \to u_2 \to u_2) \to u_2 \to u_1\ list \to u_2$ and we can type

let `fold` $= e_1$ **in**
`(fold cons nil [5, 7, 9]; fold cons nil [true, true, false])`

because the let rule for non-expansive let expressions allows us to generalise on u_1 and u_2 in the type of `fold`.

However, we will *not* be able to type the very similar

let `fold` $= e_1$ **in**
let `fast_reverse = fold cons nil` **in**
`(fast_reverse [5, 7, 9]; fast_reverse [true, true, false])`

because `fold cons nil` somewhat unjustly will be deemed expansive so that `fast_reverse` cannot get the polymorphic type $\forall u.u\ list \to u\ list$.

Fortunately there is an easy way of making it syntactically obvious that an expression does not create any new references: turn it into a lambda abstraction. This idea can be used whenever a curried function is partially applied to give a new function, and it can be used in other situations as well—although one has to make sure, of course, that stopping evaluation with an abstraction does not change the meaning of the program. In the case of `fold`, we simply change the definition of `fast_reverse` to

let `fast_reverse = fn l =>fold cons nil l` **in** $\cdots$

and `fast_reverse` is once again a polymorphic function.

3.3. *A Type Checker*

Figure 5 contains a type checker for the imperative type discipline. The algorithm is called W_1 because of its close similarity to the algorithm W in (Damas and Milner, 1982). W_1 takes as arguments an expression e and a type environment TE and returns either **fail** or a pair (S, τ) of a substitution and a type such that $S(TE) \vdash e \Rightarrow \tau$. Moreover, when W_1 succeeds, the type scheme $\sigma = \mathrm{Clos}_{\{\}}\tau$ is PRINCIPAL FOR e IN $S(TE)$ meaning that for all types τ', if $S(TE) \vdash e \Rightarrow \tau'$ then $\sigma \succ \tau'$. Finally, **fail** is returned only in

$W_1(TE, e) =$ case e of

$x \quad \Rightarrow$ if $x \notin \mathrm{Dom}\ TE$ then **fail**
else let $\forall\alpha_1 \cdots \alpha_n.\tau = TE(x)$
$\beta_1, ..., \beta_n$ be new such that
α_i is applicative iff β_i is applicative
in $(ID, \{\alpha_i \mapsto \beta_i\}\ \tau)$

fn $x \Rightarrow e_1 \Rightarrow$ let t be a new applicative type variable
$(S_1, \tau_1) = W_1(TE + \{x \mapsto t\}, e_1)$
in $(S_1, S_1(t) \to \tau_1)$

$e_1 e_2 \quad \Rightarrow$ let $(S_1, \tau_1) = W_1(TE, e_1)$;
$(S_2, \tau_2) = W_1(S_1(TE), e_2)$
t be a new applicative type variable
$S_3 = Unify_1(S_2(\tau_1), \tau_2 \to t)$ (may **fail**)
in $(S_3 S_2 S_1, S_3(t))$

let $x = e_1$ **in** $e_2 \Rightarrow$
let $(S_1, \tau_1) = W_1(TE, e_1)$
$\sigma =$ if e_1 is non-expansive then $\mathrm{Clos}_{S_1 TE}\tau_1$
else $\mathrm{AppClos}_{S_1 TE}\tau_1$
$(S_2, \tau_2) = W_1(S_1 TE + \{x \mapsto \sigma\}, e_2)$
in $(S_2 S_1, \tau_2)$

FIG. 5. A type checker for the imperative type discipline.

case there exist no (S, τ) satisfying $S(TE) \vdash e \Rightarrow \tau$. These facts have been proved, but the proofs are too long to be included in this paper.

W_1 uses a modified unification algorithm, $Unify_1$, which is like ordinary unification, except that

$$Unify_1(\alpha, \tau) = \begin{cases} \{\alpha \mapsto \tau\}, & \text{if } \alpha \text{ is applicative;} \\ \{\alpha \mapsto S(\tau)\} \cup S, & \text{if } \alpha \text{ is imperative} \end{cases}$$

provided α does not occur in τ, where $\{t_1, ..., t_n\}$ is the set of applicative type variables occurring in τ, $\{u_1, ..., u_n\}$ are new imperative type variables, and S is $\{t_1 \mapsto u_1, ..., t_n \mapsto u_n\}$.

4. Proof of Soundness

We shall now prove the soundness of the imperative type inference system. Substitutions are at the core of all we do, so we start by proving lemmas about substitutions and type inference. Then we shall define the quaternary relation $s: ST \models v: \tau$ (discussed in Section 2) as the maximal fixed point of a monotonic operator. We review the principle of co-induction and use it to prove two lemmas about the $\models$ relation. Finally, we state and prove the main soundness result.

4.1. *Lemmas about Substitutions*

More notation about maps: Let f be any map. $\mathrm{Rng}(f)$ means the range of f and $f\downarrow A$ means the restriction of f to A. When $\mathrm{Dom}(f)\cap \mathrm{Dom}(g)=\varnothing$ we write $f \mid g$ for $f+g$. We say that $f \mid g$ is THE SIMULTANEOUS COMPOSITION of f and g. Note that for every $a\in \mathrm{Dom}(f \mid g)$ we have that either $(f \mid g)\, a = f(a)$ or $(f \mid g)\, a = g(a)$. We say that g EXTENDS f, written $f\subseteq g$ if $\mathrm{Dom}(f)\subseteq \mathrm{Dom}(g)$ and for all x in the domain of f we have $f(x)=g(x)$.

The REGION of a (normally finite) substitution is defined by

$$\mathrm{Reg}(S)=\bigcup_{\alpha\in \mathrm{Dom}(S)} tyvars(S(\alpha)).$$

Substitution on type schemes and type environments is not a function because of the need for renaming of bound type variables. Instead, we define substitutions on type schemes and type environments by ternary relations $\sigma_1 \to^S \sigma_2$ and $TE \to^S TE'$:

DEFINITION 4.1. *Let* $\sigma_1=\forall\alpha_1\cdots\alpha_n.\tau_1$ *and* $\sigma_2=\forall\beta_1\cdots\beta_m.\tau_2$ *be type schemes and* S *be a substitution. We write* $\sigma_1\to^S\sigma_2$ *if* $m=n$, *and* $\{\alpha_i\mapsto\beta_i \mid 1\leqslant i\leqslant n\}$ *is a bijection and* α_i *is imperative* iff β_i *is imperative, and no* β_i *is in* $\mathrm{Reg}(S_0)$, *and* $(S_0\mid\{\alpha_i\mapsto\beta_i\})\,\tau_1=\tau_2$, *where* $S_0=^{\mathrm{def}} S\downarrow tyvars\ \sigma_1$. *Moreover, we write* $TE\to^S TE'$ *if* Dom $TE=$ Dom TE' *and for all* $x\in$ Dom TE, $TE(x)\to^S TE'(x)$.

We write $\sigma_1=_\alpha\sigma_2$ as a shorthand for $\sigma_1\to^{ID}\sigma_2$. Note that this is the familiar notion of α-conversion. One can prove that if $\sigma\succ\tau$ and $\sigma\to^S\sigma'$ then $\sigma'\succ S\tau$. The following lemma will be used again and again in what follows.

LEMMA 4.2. *If* $TE\vdash e\Rightarrow\tau$ *and* $TE\to^S TE'$ *then* $TE'\vdash e\Rightarrow S\tau$.

Proof. By structural induction on e. The only interesting case is the one for $e=$ **let** $x=e_1$ **in** e_2 which in turn is proved by case analysis. (The two cases are similar, but there are subtle differences and since this lemma is terribly important, we had better be careful here).

e_1 *is non-expansive.* Here $TE\vdash e\Rightarrow\tau$ was inferred by

$$\frac{e_1 \text{ is non-expansive} \quad TE\vdash e_1\Rightarrow\tau_1 \quad TE+\{x\mapsto \mathrm{Clos}_{TE}\tau_1\}\vdash e_2\Rightarrow\tau}{TE\vdash \textbf{let } x=e_1 \textbf{ in } e_2\Rightarrow\tau}. \quad (21)$$

It will not do simply to apply induction on e_1 using the premise $TE\vdash e_1\Rightarrow\tau_1$ and the substitution S itself, for S may act upon the type variables in τ_1 that are not free in TE. Let $\sigma_1=\mathrm{Clos}_{TE}\tau_1=\forall\alpha_1\cdots\alpha_n.\tau_1$ and

let $S_1 = S \downarrow tyvars\ TE$. Choose distinct type variables $\beta_1 \cdots \beta_n$ such that β_i is imperative iff α_i is imperative and no β_i is in Reg S_1. We then define $S' = S_1 | \{\alpha_i \mapsto \beta_i\}$. Note that $TE \to^{S'} TE'$. Therefore, applying induction to e_1 using S' for S we get

$$TE' \vdash e_1 \Rightarrow S'\tau_1. \tag{22}$$

Thus we are interested in $\text{Clos}_{TE'} S'\tau_1$. Let $S_0 = S \downarrow tyvars\ \sigma_1$. Then no β_i is in Reg S_0, so

$$\begin{aligned}
\text{Clos}_{TE}\tau_1 = \forall\alpha_1 \cdots \alpha_n.\tau_1 \xrightarrow{S} \forall\beta_1 \cdots \beta_n.(S_0 | \{\alpha_i \mapsto \beta_i\})\,\tau_1 \quad &\text{by Definition 4.1} \\
= \forall\beta_1 \cdots \beta_n.(S_1 | \{\alpha_i \mapsto \beta_i\})\,\tau_1 \\
= \forall\beta_1 \cdots \beta_n.S'\tau_1 \\
= \text{Clos}_{TE'} S'\tau_1.
\end{aligned}$$

The last of these equations is seen as follows. No β_i is free in TE' since $TE \to^S TE'$. Conversely, any type variable that is not a β_i but occurs in $S'\tau_1$ must be free in TE'; the reason for this is that every type variable free in σ_1 is in TE and $TE \to^S TE'$.

Thus we have

$$TE + \{x \mapsto \text{Clos}_{TE}\,\tau_1\} \xrightarrow{S} TE' + \{x \mapsto \text{Clos}_{TE'} S'\tau_1\}$$

so by induction on e_2, using the third premise of (21), this time with S itself, we get

$$TE' + \{x \mapsto \text{Clos}_{TE'} S'\tau_1\} \vdash e_2 \Rightarrow S\tau. \tag{23}$$

Thus by rule (19) on (22) and (23), we have $TE' \vdash e \Rightarrow S\tau$ as desired.

e_1 is expansive. Here $TE \vdash e \Rightarrow \tau$ was inferred by

$$\frac{e_1 \text{ is expansive} \quad TE \vdash e_1 \Rightarrow \tau_1 \quad TE + \{x \mapsto \text{AppClos}_{TE}\tau_1\} \vdash e_2 \Rightarrow \tau}{TE \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \Rightarrow \tau}. \tag{24}$$

This case is similar, but now S must be given the chance to act on the imperative type variables of τ_1 that are not free in TE. Thus, let $\sigma_1 = \text{AppClos}_{TE}\tau_1 = \forall\alpha_1 \cdots \alpha_n.\tau_1$ and let $S_1 = S \downarrow (tyvars\ TE \cup imptyvars\ \tau_1)$. Every α_i is applicative. Let $\beta_1 \cdots \beta_n$ be distinct applicative type variables none of which is in Reg S_1 and let $S' = S_1 | \{\alpha_i \mapsto \beta_i\}$. Note that $TE \to^{S'} TE'$. Therefore, applying induction to e_1, using S' for S, we get

$$TE' \vdash e_1 \Rightarrow S'\tau_1. \tag{25}$$

Now we are interested in $\text{AppClos}_{TE'} S'\tau_1$. Let $S_0 = S \downarrow \mathit{tyvars}\ \sigma_1$. Then no β_i is in Reg S_0, so

$$\begin{aligned}\text{AppClos}_{TE}\tau_1 = \forall\alpha_1 \cdots \alpha_n.\tau_1 &\xrightarrow{S} \forall\beta_1 \cdots \beta_n.(S_0 \mid \{\alpha_i \mapsto \beta_i\})\,\tau_1 \\ &\quad \text{by Definition 4.1} \\ &= \forall\beta_1 \cdots \beta_n.(S_1 \mid \{\alpha_i \mapsto \beta_i\})\,\tau_1 \\ &= \forall\beta_1 \cdots \beta_n.S'\tau_1 \\ &= \text{AppClos}_{TE'}\, S'\tau_1.\end{aligned}$$

The last of these equations is seen as follows. No β_i is free in TE', since $TE \to^S TE'$. Conversely, any applicative type variable that is not a β_i but occurs in $S'\tau_1$ must be free in TE'; the reasons for this are

1. Any applicative α in τ_1 which is not an α_i is free in TE and $TE \to^S TE'$.
2. Any imperative α in τ_1 is mapped by S to an imperative type, i.e., a type with no applicative type variables.

Having established the above equations, we get

$$TE + \{x \mapsto \text{AppClos}_{TE}\tau_1\} \xrightarrow{S} TE' + \{x \mapsto \text{AppClos}_{TE'}\, S'\tau_1\}$$

so by induction on e_2, using the third premise of (24), we get

$$TE' + \{x \mapsto \text{AppClos}_{TE'} S'\tau_1\} \vdash e_2 \Rightarrow S\tau$$

which with (25) gives the desired $TE \vdash e \Rightarrow S\tau$ by rule (20). ∎

The following lemma will be used in the proof of Lemma 4.7.

LEMMA 4.3. *Assume $\sigma \to^S \sigma'$ and $\sigma' \succ \tau_1'$ and A is a set of type variables with tyvars $\sigma \subseteq A$. Then there exists a type τ_1 and a substitution S_1 such that $\sigma \succ \tau_1$, $S_1\tau_1 = \tau_1'$, and $S \downarrow A = S_1 \downarrow A$.*

For a proof, see (Tofte, 1988, pp. 50–51).

4.2. *Typing of Values Using Maximal Fixed Points*

Recall from the discussion in Section 2 that the relation between dynamic values v and types τ depends not just on the store, but also on a store typing. We introduced the notion of imperative type to be able to recognise types that are types of stored values. Hence a STORE TYPING is a

map from addresses to imperative types; a TYPED STORE is a store and a store typing with equal domains:

$$ST \in \text{StoreTyping} = \text{Addr} \xrightarrow{\text{fin}} \text{ImpType}$$

$$s : ST \in \text{TypedStore} = \{(s, ST) \in \text{Store} \times \text{StoreTyping} \mid \text{Dom } s = \text{Dom } ST\}.$$

Notice that there are no bound type variables in store typings. Before proceeding with the formal development, let us briefly consider what happens if one allows general type schemes in store typings. The purpose of such an extension would be to make it possible to store and retrieve polymorphic values, for instance, polymorphic functions, without impairing their polymorphic status. As references are values, we would have to admit type expressions of the form σ *ref*, where σ is a type scheme. Notice that since such a type can occur inside a larger type, which in turn can be quantified, we have hereby introduced nested quantification in types. The rule for assignment should be, loosely speaking, that a value can be stored in a reference only if the type scheme of the value is at least as general as the type scheme of the reference. However, it is no longer clear what "more general" means. Consider, for example, the two types $(\forall t.t \to t)$ *ref* and $(int \to int)$ *ref*. Neither is in all respects more general than the other; references of the latter type can hold more functions (making assignments easier) whereas references of the former type can hold functions that are more general (so that the contents of the reference can be used in more situations). Thus, there is no natural candidate for a principal type of the expression "`ref(`**fn** x => x`)`." Indeed, the usual method of solving type equations using first-order unification breaks down when type variables range over types that can contain quantified type variables. To avoid having to tackle these very hard problems, which seem to stem from nested quantification rather than from the imperative language features, we limit ourselves to the unquantified version, whose soundness in itself is far from obvious.

We now return to the definition of the relation $s : ST \models v : \tau$. On basic values it extends the basic relation $\text{IsOf} \subseteq \text{BasVal} \times \text{TyCon}$, which contains for instance, (3, *int*), (*true*, *bool*), and (*done*, *stm*). We assume that that *done* is the only basic value which is of type *stm*. We wish to define a relation with the following property.

Property 4.4 (of $\models$). We have

$s : ST \models v : \tau \Leftrightarrow$

if $v = b$ then v IsOf τ;

if $v = [x, e_1, E]$ then there exists a TE such that $TE \vdash \mathbf{fn}\ x \Rightarrow e_1 \Rightarrow \tau$ and

$s: ST \models E: TE$, where $s: ST \models E: TE$ is short for

Dom E = Dom TE and $\forall x \in$ Dom $TE\ \forall \tau' \prec TE(x)\ s: ST \models E(x): \tau'$;

if $v = asg$ then $\tau = \tau_1\ ref \to \tau_1 \to stm$ for some τ_1;

if $v = ref$ then $\tau = \theta \to \theta\ ref$ for some imperative θ;

if $v = deref$ then $\tau = \tau_1\ ref \to \tau_1$ for some τ_1;

if $v = a$ then $\tau = (ST(a))\ ref$ and $s: ST \models s(a): ST(a)$.

The above property does not define a unique relation $\models$. However, it can be regarded as a fixed point equation

$$\models = F(\models), \tag{26}$$

where F is an operator defined as follows. Let $U =$ TypedStore $\times$ Val $\times$ Type and let $P(U)$ denote the set of subsets of U. Then $F: P(U) \to P(U)$ is defined by

$F(Q) = \{(s: ST, v, \tau) |$

if $v = b$ then v IsOf τ;

if $v = [x, e_1, E]$ then there exists a TE such that $TE \vdash \mathbf{fn}\ x \Rightarrow e_1 \Rightarrow \tau$ and

Dom E = Dom TE and $\forall x \in$ Dom $TE\ \forall \tau' \prec TE(x)\ (s: ST, E(x), \tau') \in Q$;

if $v = asg$ then $\tau = \tau_1\ ref \to \tau_1 \to stm$ for some τ_1;

if $v = ref$ then $\tau = \theta \to \theta\ ref$ for some imperative θ;

if $v = deref$ then $\tau = \tau_1\ ref \to \tau_1$ for some τ_1;

if $v = a$ then $\tau = (ST(a))\ ref$ and $(s: ST, s(a), ST(a)) \in Q\}$.

It is crucial that F is MONOTONIC, i.e., that $Q \subseteq Q'$ implies $F(Q) \subseteq F(Q')$. This would not have been the case, had we taken the following, perhaps more natural, definition of F:

$F(Q) = \{(s: ST, v, \tau) |$

...

if $v = [x, e_1, E]$ then $\tau = \tau_1 \to \tau_2$ and

for all v_1, v_2, s'

if $(s: ST, v_1, \tau_1) \in Q$ and $s, E + \{x \mapsto v_1\} \vdash e_1 \longrightarrow v_2, s'$

then $\exists\ ST' \supseteq ST$ such that $(s': ST', v_2, \tau_2) \in Q$

... $\}$

However, the chosen F is monotonic, so it has a smallest and a greatest fixed point in the complete lattice $(P(U), \subseteq)$, namely

$$R^{\min} = \bigcap \{Q \subseteq U \mid F(Q) \subseteq Q\}$$

and

$$R^{\max} = \bigcup \{Q \subseteq U \mid Q \subseteq F(Q)\}. \tag{27}$$

For our particular F, the minimal fixed point $R^{\min}$ is strictly contained in the maximal fixed point $R^{\max}$ and it turns out that it is the latter we want. This is due to the possibility of cycles in the store as illustrated by the following example.

EXAMPLE 4.5. Consider the evaluation of

```
let r = ref(fn x => x + 1)
in let s = ref(fn y => (!r) y + 2)
  in r := !s
```

in the empty store. At the point just before "`r := !s`" is evaluated, the store appears as

$$\{\ a_1 \mapsto [\mathrm{x}, \mathrm{x}+1, E_0],\quad a_2 \mapsto [\mathrm{y}, (!\mathrm{r})\mathrm{y}+2, E_0 + \{\mathrm{r} \mapsto a_1\}]\ \},$$

where E_0 is the initial environment. After the assignment the store becomes cyclic:

$$s' = \{\ a_1 \mapsto [\mathrm{y}, (!\mathrm{r})\mathrm{y}+2, E_0 + \{\mathrm{r} \mapsto a_1\}],\quad a_2 \mapsto [\mathrm{y}, (!\mathrm{r})\mathrm{y}+2, E_0 + \{\mathrm{r} \mapsto a_1\}]\ \}.$$

Now we would expect to have $s' : ST' \models a_1 : (int \to int)\ ref$, where

$$ST' = \{a_1 \mapsto int \to int,\ a_2 \mapsto int \to int\}.$$

Indeed, if we let $q = (s' : ST', a_1, (int \to int)\ ref)$ then we do have $q \in R^{\max}$. To prove this it will suffice to find a Q with $q \in Q$ and $Q \subseteq F(Q)$, since we have (27). But it is easy to check that $Q = \{(s' : ST', a_1, (int \to int)\ ref), (s' : ST', [\mathrm{y}, (!\mathrm{r})\mathrm{y}+2, \{\mathrm{r} \mapsto a_1\}],\ int \to int)\}$ satisfies $Q \subseteq F(Q)$. As we shall see below, one can think of this Q as the *smallest consistent* set of typings containing q.

On the other hand, q is not in $R^{\min}$. This can be seen as follows. There is an alternative characterisation of $R^{\min}$, namely

$$R^{\min} = \bigcup_{\lambda} F^{\lambda}, \tag{28}$$

where $F^{\lambda} = F(\bigcup_{\mu<\lambda} F^{\mu})$, where λ ranges over all ordinals (see Aczel, 1977, for an introduction to inductive definitions). In other words, one obtains $R^{\min}$ by starting from the empty set and then applying F iteratively. It is easy to show that because q is cyclic there is no least ordinal λ such that $q \in F^{\lambda}$. Therefore $q \notin R^{\min}$.

The distinction between minimal and maximal fixed points in operational semantics is treated in some detail in (Milner and Tofte, 1990; Tofte, 1988). See also (Aczel, 1988) for an excellent treatment of non-well-founded sets in a more mathematical setting. For any set U and for any monotonic operator $F: P(U) \to P(U)$, let us say that a set $Q \subseteq U$ is F-CONSISTENT if $Q \subseteq F(Q)$. If one thinks of Q being a set of *claims*, the use of the term "consistency" is natural. In the case at hand, Q is a set of claims, each claim being of the form $(s: ST, v, \tau)$ claiming that v has type τ in $s: ST$; moreover, Q is F-consistent if for every $q \in Q$, q is in $F(Q)$, where $F(Q)$ is the set of claims which F admits on the basis that the claims in Q are taken for granted. Notice that it is consistent to claim that a_1 is an $(int \to int)$ *ref*, although it cannot be proved constructively, starting from the empty set of claims.

In general, from (27) we see that $R^{\max}$ contains any F-consistent set. Thus we get the principle of CO-INDUCTION:

> *Let U be any set, let $F: P(U) \to P(U)$ be a monotonic function and let R be the maximal fixed point of F. For any $Q \subseteq U$, in order to prove $Q \subseteq R$, it is sufficient to prove that Q is F-consistent i.e., that $Q \subseteq F(Q)$.*

To sum up, we define that $s: ST \models v: \tau$ if $(s: ST, v, \tau) \in R^{\max}$, that $s: ST \models v: \sigma$ if for all $\tau \prec \sigma$ we have $s: ST \models v: \tau$, and that $s: ST \models E: TE$ if $\mathrm{Dom}\, E = \mathrm{Dom}\, TE$ and for all $x \in \mathrm{Dom}\, E$, $s: ST \models E(x) : TE(x)$. The aim of the remainder of this section is to prove the final soundness proposition of Section 2 for the imperative type discipline and the above definition of the $\models$ relation.

4.3. *Proofs Using Co-induction*

For the soundness proof we need to prove certain properties of the $\models$ relation. The following two lemmas are proved using co-induction; as this proof technique is less common in semantics than structural induction

and proof by the depth of inference, we wish to document the two proofs in some detail.

A typed store $s: ST'$ is said to SUCCEED a typed store $s: ST$, written $s: ST \sqsubseteq s': ST'$, if $ST \subseteq ST'$ and for all v, τ, if $s: ST \models v: \tau$ then $s': ST' \models v: \tau$. (As usual, the notation $ST \subseteq ST'$ means $\mathrm{Dom}\, ST \subseteq \mathrm{Dom}\, ST'$ and for all $x \in \mathrm{Dom}\, ST$, $ST(x) = ST'(x)$.) The relation $\sqsubseteq$ is obviously reflexive and transitive. It is not antisymmetric. Notice that if $s: ST \sqsubseteq s': ST'$ and $\mathrm{Dom}\, s = \mathrm{Dom}\, s'$ then $ST = ST'$, since $\mathrm{Dom}\, ST = \mathrm{Dom}\, s = \mathrm{Dom}\, s' = \mathrm{Dom}\, ST'$.

The first lemma concerns the creation of a new reference ($a_0 \notin \mathrm{Dom}\, ST$) and assignment ($ST(a_0) = \theta$). As before, θ ranges over imperative types.

LEMMA 4.6 (Side effects). *If* $s: ST \models v_0: \theta$ *and either* $a_0 \notin \mathrm{Dom}\, ST$ *or* $ST(a_0) = \theta$ *then* $s: ST \sqsubseteq s + \{a_0 \mapsto v_0\}: ST + \{a_0 \mapsto \theta\}$.

This lemma crucially depends on the $\models$ relation being the *maximal* fixed point of F. Had we chosen the minimal fixed point, the lemma would not hold, for, as illustrated by Example 4.5, it is precisely using assignments that one can turn a well-founded store into a non-well-founded store.

Proof. It will suffice to prove that for all v, τ, if $s: ST \models v: \tau$ then $s': ST' \models v: \tau$, where $s' = s + \{a_0 \mapsto v_0\}$ and $ST' = ST + \{a_0 \mapsto \theta\}$. This is proved by co-induction. Let

$$Q = \{(s': ST', v, \tau) \mid s: ST \models v: \tau\},$$

where $s: ST$, v_0, θ and a_0 are given and satisfy $s: ST \models v_0: \theta$ and $a_0 \notin \mathrm{Dom}\, ST$ or $ST(a_0) = \theta$ and s' is $s + \{a_0 \mapsto v_0\}$ and ST' is $ST + \{a_0 \mapsto \theta\}$. By co-induction, it will suffice to prove that Q is F-consistent. So take $q = (s': ST', v, \tau) \in Q$; then

$$s: ST \models v: \tau. \tag{29}$$

To establish $q \in F(Q)$ we proceed by case analysis.

If $v = b$ then v IsOf τ by Property 4.4 on (29). Thus $q \in F(Q)$ as desired. Similarly for $v = asg$, *ref*, and *deref*.

If $v = [x, e_1, E]$ then by the Property of $\models$ on (29) there exists a TE such that $TE \vdash \mathbf{fn}\ x \Rightarrow e_1 \Rightarrow \tau$ and $\mathrm{Dom}\, E = \mathrm{Dom}\, TE$ and for all $x \in \mathrm{Dom}\, TE$ and for all $\tau' \prec TE(x)$, $s: ST \models E(x): \tau'$. But $s: ST \models E(x): \tau'$ implies $(s': ST', E(x), \tau') \in Q$. Thus $q \in F(Q)$.

If $v = a$ then $\tau = (ST(a))\ ref$ and $s: ST \models s(a): ST(a)$ by Property 4.4 on (29). Since $ST \subseteq ST'$ we therefore have $\tau = (ST'(a))\ ref$. Moreover, since $s: ST \models s(a): ST(a)$ and $s: ST \models v_0: \theta$, we have $s: ST \models s'(a): ST'(a)$. Thus $(s': ST', s'(a), ST'(a)) \in Q$. This with $\tau = (ST'(a))\ ref$ gives $(s': ST', a, \tau) \in F(Q)$ i.e., $q \in F(Q)$. ∎

The second lemma is crucial in the case regarding the let rule. In general, there can be many consistent choices of ST and τ for given s and v all satisfying $s: ST \models v: \tau$, but the lemma says that if one choice of ST and τ is consistent, then so is any substitution instance. In that sense, an imperative type variables occurring in ST can be regarded as standing for a fixed, but unknown, monotype.

LEMMA 4.7 (Semantic substitution). *If* $s: ST \models v: \tau$ *then* $s: S(ST) \models v: S(\tau)$ *for all substitutions* S.

Proof. The proof is done by co-induction, since the $\models$ relation is non-well-founded. Define

$$Q = \{(s: ST', v, \tau') \mid \exists S, \tau \text{ s.t. } S(ST) = ST' \wedge S(\tau) = \tau' \wedge s: ST \models v: \tau\},$$

where s, ST, and ST' are given. By co-induction it will suffice to prove that Q is F-consistent. So take $q = (s: ST', v, \tau') \in Q$. Let S and τ be such that

$$S(ST) = ST' \text{ and } S(\tau) = \tau' \text{ and } s: ST \models v: \tau. \tag{30}$$

To establish $q \in F(Q)$ we proceed by case analysis.

If $v = b$ then v IsOf τ by Property 4.4 on (30). Thus $\tau \in \text{TyCon}$, so $\tau' = \tau$. Thus $q \in F(Q)$.

If $v = asg$, then by (30) we have $\tau = \tau_1\ ref \rightarrow (\tau_1 \rightarrow stm)$ for some τ_1. Thus $\tau' = (S\tau_1)\ ref \rightarrow (S\tau_1 \rightarrow stm)$ showing $q \in F(Q)$. Similarly for $v = deref$.

If $v = ref$ then $\tau = \theta \rightarrow \theta\ ref$ for some imperative type θ. Since substitutions are required to map imperative type variables to imperative types, we have that $S(\theta)$ is an imperative type. Thus $\tau' = S\theta \rightarrow (S\theta)\ ref$ showing $q \in F(Q)$.

If $v = [x, e_1, E]$ then by (30) there exists a TE such that $TE \vdash \mathbf{fn}\ x \Rightarrow e_1 \Rightarrow \tau$ and Dom E = Dom TE and

$$\forall x \in \text{Dom } TE\ \forall \tau_1 \prec TE(x)\ s: ST \models E(x): \tau_1. \tag{31}$$

There exists a TE' such that $TE \rightarrow^S TE'$. We shall now see that this TE' suffices for the TE occurring in the definition of F. By Lemma 4.2 we have $TE' \vdash \mathbf{fn}\ x \Rightarrow e_1 \Rightarrow S(\tau)$, i.e., $TE' \vdash \mathbf{fn}\ x \Rightarrow e_1 \Rightarrow \tau'$ as desired. Moreover, Dom E = Dom TE'. Finally, still following the definition of F, take $x \in \text{Dom } TE'$ and $\tau_1' \prec TE'(x)$. Let $A = tyvars(ST) \cup tyvars\ TE(x)$. We have $TE(x) \rightarrow^S TE'(x)$ and $TE'(x) \succ \tau_1'$. Then by Lemma 4.3 there exists a τ_1 and a substitution S_1 such that $TE(x) \succ \tau_1$, $S_1\tau_1 = \tau_1'$ and $S_1 \downarrow A = S \downarrow A$. In particular, $S_1(ST) = ST'$.

From (31) we get $s: ST \models E(x): \tau_1$. From $S_1(ST) = ST'$ and $S_1\tau_1 = \tau_1'$

and $s: ST \models E(x): \tau_1$ we get $(s: ST', E(x), \tau_1') \in Q$. Thus TE' satisfies the requirements in the definition of F, proving that $q \in F(Q)$ in this final case. ▮

4.4. *The Consistency Theorem*

THEOREM 4.8 (Consistency of static and dynamic semantics). *If* $s: ST \models E: TE$ *and* $TE \vdash e \Rightarrow \tau$ *and* $s, E \vdash e \longrightarrow v, s'$ *then there exists an* ST' *with* $s: ST \sqsubseteq s': ST'$ *and* $s': ST' \models v: \tau$.

This clearly implies the final soundness proposition of Section 2. It also implies the first soundness proposition. Hence the theorem ensures that if e elaborates to a basic type π and evaluates to a basic value b then b IsOf π.

Proof (of Theorem 4.8). The proof is by induction on the depth of the dynamic evaluation. There is one case for each rule. The cases concerning a variable (rule (1)) and a lambda abstraction (rule (2)) are straightforward. In the remaining cases there are always more than one premise in the evaluation rule. Here the "$s: ST \sqsubseteq s': ST'$" in the induction hypothesis is crucial; for it implies that for all v and τ, if $s: ST \models v: \tau$ then $s': ST' \models v: \tau$. In particular, $s: ST \sqsubseteq s': ST'$ and $s: ST \models E: TE$ implies $s': ST' \models E: TE$, so the assumption that the dynamic environment matches the type environment can be carried through the individual steps of the evaluation. With these comments, most of the cases are routine inductive arguments; readers who feel that their patience is being stretched may proceed to the case concerning let expressions.

Application of a closure, rule (3). Here the situation is

$$\frac{TE \vdash e_1 \Rightarrow \tau' \to \tau \quad TE \vdash e_2 \Rightarrow \tau'}{TE \vdash e_1 e_2 \Rightarrow \tau} \tag{32}$$

and

$$\frac{\begin{array}{l} s, E \vdash e_1 \longrightarrow [x_0, e_0, E_0], s_1 \\ s_1, E \vdash e_2 \longrightarrow v_2, s_2 \\ s_2, E_0 + \{x_0 \mapsto v_2\} \vdash e_0 \longrightarrow v, s' \end{array}}{s, E \vdash e_1 e_2 \longrightarrow v, s'}. \tag{33}$$

By induction on the first premises of (32) and (33) there exists an ST_1 such that $s: ST \sqsubseteq s_1: ST_1$ and

$$s_1: ST_1 \models [x_0, e_0, E_0]: \tau' \to \tau. \tag{34}$$

Before we can apply induction a second time, we must establish $s_1: ST_1 \models E: TE$; but this follows from $s: ST \models E: TE$ and $s: ST \sqsubseteq s_1: ST_1$.

Applying induction a second time, this time on the second premise of (33), together with $s_1 : ST_1 \models E : TE$ and the second premise of (32), there exists an ST_2 such that $s_1 : ST_1 \sqsubseteq s_2 : ST_2$ and

$$s_2 : ST_2 \sqsubseteq v_2 : \tau'. \tag{35}$$

Now (34) together with $s_1 : ST_1 \sqsubseteq s_2 : ST_2$ gives

$$s_2 : ST_2 \models [x_0, e_0, E_0] : \tau' \to \tau.$$

Thus by Property 4.4 there exists a TE_0 such that

$$s_2 : ST_2 \sqsubseteq E_0 ; TE_0 \tag{36}$$

and

$$TE_0 \vdash \mathbf{fn}\, x_0 \Rightarrow e_0 \Rightarrow \tau' \to \tau. \tag{37}$$

But (37) must be due to

$$TE_0 + \{x_0 \mapsto \tau'\} \vdash e_0 \Rightarrow \tau. \tag{38}$$

From (36) and (35) we get

$$s_2 : ST_2 \models E_0 + \{x_0 \mapsto v_2\} : TE_0 + \{x_0 \mapsto \tau'\}. \tag{39}$$

Thus we can apply induction a third time, this time to the third premise of (33) together with (39) and (39), to get an ST' such that $s_2 : ST_2 \sqsubseteq s' : ST'$ and the desired $s' : ST' \models v : \tau$. Also, the desired $s : ST \sqsubseteq s' : ST'$ follows from the transitivity of $\sqsubseteq$.

Notice that we could not have done induction on the depth of the type inference as we do not know anything about the depth of (37). Also note that the present definition of what it is for a closure to have a type (which almost was forced upon us because we needed F to be monotonic) now most conveniently provides the TE_0 for (36).

Assignment, rule (4). This is the first case where the lemma concerning side-effects is used. We have $e = (e_1 e_2)\, e_3$ so the inferences must have been

$$\frac{TE \vdash e_1 \Rightarrow \tau'' \to (\tau' \to \tau) \quad TE \vdash e_2 \Rightarrow \tau''}{TE \vdash e_1 e_2 \Rightarrow \tau' \to \tau} \tag{40}$$

$$\frac{TE \vdash e_1 e_2 \Rightarrow \tau' \to \tau \quad TE \vdash e_3 \Rightarrow \tau'}{TE \vdash (e_1 e_2)\, e_3 \Rightarrow \tau} \tag{41}$$

$$\frac{\begin{array}{c} s, E \vdash e_1 \longrightarrow asg, s_1 \\ s_1, E \vdash e_2 \longrightarrow a, s_2 \\ s_2, E \vdash e_3 \longrightarrow v_3, s_3 \end{array}}{s, E \vdash (e_1 e_2)\, e_3 \longrightarrow done, s_3 + \{a \mapsto v_3\}}, \tag{42}$$

where $s' = s_3 + \{a \mapsto v_3\}$.

By induction on the first premise of (40) and (42) there exists a ST_1 such that $s: ST \sqsubseteq s_1: ST_1$ and $s_1: ST_1 \models asg: \tau'' \to (\tau' \to \tau)$. By Property 4.4 we must have $\tau'' = \tau'$ *ref* and $\tau = stm$.

Now $s_1: ST_1 \models E: TE$. By induction on the second premises of (40) and (42) we therefore get a ST_2 such that $s_1: ST_1 \sqsubseteq s_2: ST_2$ and $s_2: ST_2 \models a: \tau'$ *ref*.

Thus $s_2: ST_2 \models E: TE$. By induction on the second premise of (41) and the third premise of (42) there exists an ST' such that $s_2: ST_2 \sqsubseteq s_3: ST'$ and $s_3: ST' \models v_3: \tau'$. In particular, we have $s_3: ST' \models a: \tau'$ *ref*. Thus $ST'(a) = \tau'$, so τ' must must be imperative and Lemma 4.6 gives $s_3: ST' \sqsubseteq s_3 + \{a \mapsto v_3\}: ST' + \{a \mapsto \tau'\} = s': ST'$. Since $(done, stm) \in \text{IsOf}$ we have $s': ST' \models done: stm$, i.e., the desired $s': ST' \models v: \tau$.

Creation of a reference, rule (5). This is the second case where the lemma concerning side-effects is used. The rules are

$$\frac{TE \vdash e_1 \Rightarrow \tau' \to \tau \quad TE \vdash e_2 \Rightarrow \tau'}{TE \vdash e_1 e_2 \Rightarrow \tau}$$

$$\frac{s, E \vdash e_1 \longrightarrow ref, s_1 \quad s_1, E \vdash e_2 \longrightarrow v_2, s_2, \quad a \notin \text{Dom } s_2}{s, E \vdash e_1 e_2 \longrightarrow a, s_2 + \{a \mapsto v_2\}}$$

where $s' = s_2 + \{a \mapsto v_2\}$. By induction on the first premises there exists a ST_1 such that $s: ST \sqsubseteq s_1: ST_1$ and $s_1: ST_1 \models ref: \tau' \to \tau$. Thus by Property 4.4 we have $\tau = \tau'$ *ref* and τ and τ' are imperative types.

Now $s_1: ST_1 \models E: TE$. Thus induction on the second premises gives an ST_2 such that $s_1: ST_1 \sqsubseteq s_2: ST_2$ and $s_2: ST_2 \models v_2: \tau'$.

Let $ST' = ST_2 + \{a \mapsto \tau'\}$. This makes sense since τ' is an imperative type. Since $a \notin \text{Dom } s_2$ and $\text{Dom } s_2 = \text{Dom } ST_2$, we have $a \notin \text{Dom } ST_2$. Since $s_2: ST_2 \models v_2: \tau'$ and τ' is imperative, Lemma 4.6 gives $s_2: ST_2 \sqsubseteq s_2 + \{a \mapsto v_2\}: ST_2 + \{a \mapsto \tau'\} = s': ST'$ as desired. Hence $s': ST' \models v_2: \tau'$, i.e., $s': ST' \models s'(a): ST'(a)$ so $s': ST' \models a: \tau'$ *ref*, i.e., $s': ST' \models v: \tau$.

Dereferencing, rule (6). Here

$$\frac{TE \vdash e_1 \Rightarrow \tau' \to \tau \quad TE \vdash e_2 \Rightarrow \tau'}{TE \vdash e_1 e_2 \Rightarrow \tau}$$

$$\frac{s, E \vdash e_1 \longrightarrow deref, s_1 \quad s_1, E \vdash e_2 \longrightarrow a, s' \quad s'(a) = v}{s, E \vdash e_1 e_2 \longrightarrow v, s'}.$$

By induction of the first premises there exists an ST_1 such that $s: ST \sqsubseteq s_1: ST_1$ and $s_1: ST_1 \models deref: \tau' \to \tau$. Thus $\tau' = \tau$ *ref*.

Now $s_1: ST_1 \models E: TE$. Thus by induction on the second premises there is an ST' such that $s_1: ST_1 \sqsubseteq s': ST'$ and $s': ST' \models a: \tau$ *ref*. Thus $s: ST \sqsubseteq s': ST'$ and $s': ST' \models s'(a): \tau$, *i.e.*, $s': ST' \models v: \tau$.

Let expressions, rule (7). The dynamic evaluation is

$$\frac{s, E \vdash e_1 \longrightarrow v_1, s_1 \quad s_1, E + \{x \mapsto v_1\} \vdash e_2 \longrightarrow v, s'}{s, E \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \longrightarrow v, s'}. \tag{43}$$

Now there are two subcases:

e_1 is expansive. Then $TE \vdash e \Rightarrow \tau$ must have been inferred by

$$TE \vdash e_1 \Rightarrow \tau_1 \tag{44}$$

and

$$TE + \{x \mapsto \mathrm{AppClos}_{TE}\tau_1\} \vdash e_2 \Rightarrow \tau \tag{45}$$

for some τ_1, by rule (20). By induction on the first premise of (43) and (44) there exists an ST_1 such that $s: ST \sqsubseteq s_1: ST_1$ and

$$s_1: ST_1 \models v_1: \tau_1. \tag{46}$$

Thus

$$s_1: ST_1 \models E: TE. \tag{47}$$

Bearing in mind that we have (45), we now want to strengthen (46) to

$$s_1: ST_1 \models v_1: \mathrm{AppClos}_{TE}\tau_1. \tag{48}$$

So take any $\tau \prec \mathrm{AppClos}_{TE}\tau_1$. Any bound variable in $\mathrm{AppClos}_{TE}\tau_1$ is applicative, so it does not occur in ST_1, simply because store typings by definition cannot contain applicative type variables. Thus $\tau \prec \mathrm{AppClos}_{TE}\tau_1$ ensures the existence of a substitution S such that $S(ST_1) = ST_1$ and $S(\tau_1) = \tau$. Thus, when we apply the semantic substitution lemma, Lemma 4.7, on (46) we get

$$s_1: ST_1 \models v_1: \tau. \tag{49}$$

Since (49) holds for arbitrary $\tau \prec \mathrm{AppClos}_{TE}\tau_1$ we have proved (48). Then (47) and (48) give

$$s_1: ST_1 \models E + \{x \mapsto v_1\}: TE + \{x \mapsto \mathrm{AppClos}_{TE}\tau_1\}. \tag{50}$$

Applying induction on the second premise of (43) and (50) and (45), we get an ST' such that $(s: ST \sqsubseteq) s_1: ST_1 \sqsubseteq s': ST'$ and $s': ST' \models v: \tau$ as desired.

e_1 *is non-expansive.* Then $TE \vdash e \Rightarrow \tau$ must have been inferred from

$$TE \vdash e_1 \Rightarrow \tau_1 \tag{51}$$

$$TE + \{x \mapsto \mathrm{Clos}_{TE}\tau_1\} \vdash e_2 \Rightarrow \tau \tag{52}$$

for some τ_1 by application of rule (19).

Let $\{\alpha_1, ..., \alpha_n\} = tyvars\ \tau_1 \backslash tyvars\ TE$. Then $\mathrm{Clos}_{TE}\tau_1 = \forall\alpha_1 \cdots \alpha_n.\tau_1$. Moreover, let $\{u_1, ..., u_m\}$ be the imperative type variables among $\{\alpha_1, ..., \alpha_n\}$. Although no u_i occurs free in TE, we cannot be sure that no u_i occurs free in ST. In general, ST contains the types of all stored values, not just of the stored values presently accessible via a variable; therefore, in the elaboration $TE \vdash e_1 \Rightarrow \tau_1$ we may have chosen imperative type variables that were "fresh" with respect to the type environment but not with respect to the store typing. However, elaboration is preserved under substitution, so we can rename these imperative type variables as follows. Let $\{u'_1, ..., u'_m\}$ be imperative type variables such that $R = \{u_i \mapsto u'_i \mid 1 \leqslant i \leqslant m\}$ is a bijection and

$$\mathrm{Rng}\ R \cap tyvars\ ST = \varnothing \tag{53}$$

$$\mathrm{Rng}\ R \cap tyvars\ TE = \varnothing. \tag{54}$$

Now $TE \to^R TE$ as no u_i is free in TE, so the substitution lemma, Lemma 4.2, applied to (51) gives

$$TE \vdash e_1 \Rightarrow R\tau_1. \tag{55}$$

Moreover, $\mathrm{Clos}_{TE}\tau_1 =_\alpha \mathrm{Clos}_{TE}(R\tau_1)$ by (54) so from (52) we get

$$TE + \{x \mapsto \mathrm{Clos}_{TE}(R\tau_1)\} \vdash e_2 \Rightarrow \tau \tag{56}$$

by using Lemma 4.2 on the identity substitution. Applying induction to the first premises of (43) and (55) we get an ST_1 such that $s: ST \sqsubseteq s_1: ST_1$ and

$$s_1: ST_1 \models v_1: R\tau_1. \tag{57}$$

Since e_1 is non-expansive, we have $\mathrm{Dom}\ s = \mathrm{Dom}\ s_1$—*and this is the crucial property of non-expansive expressions.* Since $s: ST \sqsubseteq s_1: ST_1$, we have $ST_1 = ST$ (recall the definition of $\sqsubseteq$ and note that $\mathrm{Dom}\ ST = \mathrm{Dom}\ s = \mathrm{Dom}\ s_1 = \mathrm{Dom}\ ST_1$). Thus

$$s_1: ST \models E: TE \tag{58}$$

and, by (57),

$$s_1: ST \models v_1: R\tau_1. \tag{59}$$

Bearing (56) in mind we want to strengthen (59) to

$$s_1 : ST \models v_1 : \mathrm{Clos}_{TE}(R\tau_1). \tag{60}$$

So take any $\tau \prec \mathrm{Clos}_{TE}(R\tau_1)$. No variable α bound in $\mathrm{Clos}_{TE}(R\tau_1)$ can occur in ST, either because α is applicative or because of (53)—this is precisely why we do the renaming.

Hence $\tau \prec \mathrm{Clos}_{TE}(R\tau_1)$ implies the existence of a substitution S with $S(ST) = ST$ and $S(R\tau_1) = \tau$. We now apply the semantic substitution lemma, Lemma 4.7, to (59) to obtain

$$s_1 : ST \models v_1 : \tau. \tag{61}$$

Since (61) holds for every $\tau \prec \mathrm{Clos}_{TE}(R\tau_1)$, we have proved (60).

From (58) and (60) we then get

$$s_1 : ST \models E + \{x \mapsto v_1\} : TE + \{x \mapsto \mathrm{Clos}_{TE}(R\tau_1)\} \tag{62}$$

Finally we apply induction to (62), the second premise of (43), and to (56) to get the desired ST'. ∎

5. Conclusion

From the proof case concerned with non-expansive expressions in let expressions we learn that the important property of a non-expansive expression is that *it does not expand the domain of the store.* Because of the very simple way we have defined what it is for an expression to be non-expansive, non-expansive expressions will in fact leave the entire store (not just its domain) unchanged. The proof shows that this is not necessary; assignments are harmless, only creation of new references is critical. (In retrospect, this explains why the type scheme for `ref` has a bound imperative type variable, while the type schemes for := and ! are purely applicative.)

The crude syntactic distinction between non-expansive and expansive expressions can be replaced by more and more sophisticated forms of static analysis to determine whether an expression extends the domain of the store. David MacQueen has invented a type discipline whereby the binary imperative/applicative attribute of type variables is replaced by a natural number, which we can call the RANK of the type variable. An α of rank 0 in the type of an expression e ranges over a type that must be assumed to be free in the implicit store typing. An α of rank n, $1 \leqslant n \leqslant \infty$, ranges over a type which is guaranteed not to become free in the store typing as long as e is applied at most $n-1$ times as a curried function.

As far as we know (it has not been proved), all expressions admitted under our scheme are admitted under Damas' scheme, and all expressions admitted under Damas' scheme are admitted under MacQueen's scheme and both these inclusions are proper.

As an alternative to more and more complicated type disciplines, one can use the type inference system we have presented and perform the analysis of when references are created separately. This has the advantage of splitting the correctness problem into two: the correctness of the type inference system, which we have proved in this paper, and the correctness of the analysis of reference creation times, which must be proved for the technique in question.

One should keep in mind that the practical aim of type checking is not simply to admit as many programs as possible (while maintaining soundness, of course). It is also important that the underlying type inference system is simple enough that users can find out why their programs are *rejected* by the type checker, when that happens. This is why I advocate the simple rule that variables and lambda abstractions are non-expansive, all other expressions ae expansive.

ACKNOWLEDGMENTS

I thank Robin Milner warmly for his superb supervision of this work. It was he who suggested that one might need the quaternary relation $s: ST \models v: \tau$ and, most importantly, it was he who realised that it should be defined as a maximal fixed point. I am also indebted to David MacQueen for delightful discussions about polymorphic references, to Robert Harper and Kevin Mitchell for their criticism of my first attempts at getting a sound system, and to the referees for their detailed comments and constructive suggestions.

RECEIVED October 19, 1988; FINAL MANUSCRIPT RECEIVED June 23, 1989

REFERENCES

ACZEL, P. (1977), An introduction to inductive definitions, *in* "Handbook of Mathematical Logic" (J. Barwise, Ed.), North-Holland, Amsterdam.

ACZEL, P. (1988), "Non-Well-Founded Sets," CSLI Lecture Notes, No. 14, CLSI/Stanford.

DAMAS, L. (1985), "Type Assignment in Programming Languages," Ph. D. thesis CST-33-85, Department of Computer Science, University of Edinburgh.

DAMAS, L., AND MILNER, R. (1982), Principal type schemes for functional programs, *in* "Proceedings, ACM Symposium on the Principles of Programming Languages," pp. 207–212.

GORDON, M., MILNER, R., AND WADSWORTH, C. (1979), "Edinburgh LCF," Lecture Notes in Computer Science, Vol. 78, Springer-Verlag, Berlin/Heidelberg/New York.

HINDLEY, R. (1969), The principal type scheme of an object in combinatory logic, *Trans. Amer. Math. Soc.* **146**, 29–60.

LAKATOS, I. (1976), "Proofs and Refutations," Cambridge Univ. Press, Cambridge.

MILNER, R. (1978), A theory of type polymorphism in programming languages, *J. Comput. System Sci.* **17**, 348–375.

MILNER, R., TOFTE, M., AND HARPER, R. (1990), "The Definition of Standard ML," MIT Press, Cambridge, MA.

MILNER, R., AND TOFTE, M. (1990), Co-induction in relational semantics, *Theoret. Comput. Sci.*, to appear.

PLOTKIN, G. (1981), "A Structural Approach to Operational Semantics," Technical Report DAIMI-FN-19, Computer Science Department, Aarhus University, Denmark.

TOFTE, M. (1988), "Operational Semantics and Polymorphic Type Inference," Ph. D. thesis CST-52-88, Department of Computer Science, Edinburgh University.